พัลลภ เฉลิมฉัตร

ร่ายยาว GUIs : Hardware Independent

ถ้าเป็นภาพยนตร์คงเบื่อกันแล้ว แต่เรื่อง GUIs ยิ่งอ่านยิ่งมัน โดยเฉพาะในฉบับนี้พบกับ Hardware Independent จุดเด่นบน GUIs อีกตัวเลยครับ

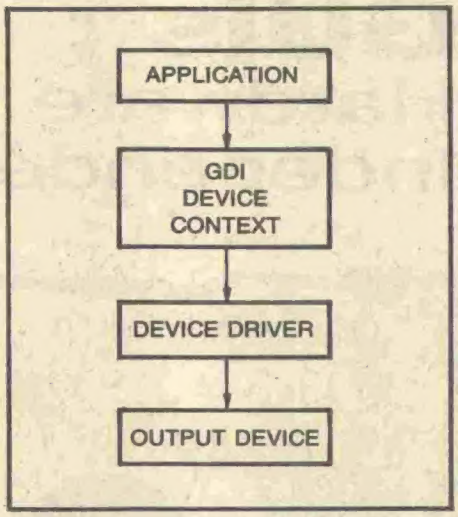
สวัสดีครับท่านผู้อ่านทุกท่าน ขณะนี้ผมกำลังพิมพ์ต้นฉบับอยู่ที่ระยองเป็นเวลา 3 วัน ที่หิ้วเจ้าแลปท็อปไปนั่งเขียนโปรแกรมและพิมพ์งานด้วย ก็นับเป็นความสะดวกสบายอีกขั้นหนึ่งที่วิวัฒนาการทางด้านฮาร์ดแวร์ส่วนทางด้านซอฟต์แวร์ก็ไม่ใช่ย่อย ตอนนี้ใคร ๆ ก็พูดถึง GUIs กันทั้งนั้นไม่ว่าจะเป็น Windows, X Window, SunView แต่คอยดูให้ดีเถอะครับในอนาคตอันใกล้เราคงมี Handwriting Notebook ขายกันเกลื่อน มีระบบติดต่อที่ใช้เสียง (Speech or Voice Recognition) ผมยังนึกไม่ออกว่าอีกห้าปีเราจะมีอะไรมาเล่นกันอีก สำหรับร่ายยาว GUIs ก็เป็นตอนที่ 3 แล้วนะครับ แต่ก็จัดอยู่ในส่วนแรกของเนื้อหาที่ผมวางไว้ ดังนั้นเพื่อไม่ให้ยืดยาวมากเกินไป ผมก็จะพยายามแทรกเนื้อหาในแต่ละตอนมากขึ้น โดยเมื่อจบเนื้อหาในขอบเขตของผมอันว่าด้วยหลักการแล้ว ก็จะเข้าสู่เทคนิคพิเศษในการเขียนโปรแกรมบน Windows โดยบรรดาเพื่อน ๆ ผมอีกทีมหนึ่ง ในตอนที่แล้วผมได้เกริ่นที่จะขึ้นหัวข้อเกี่ยวกับ message แต่เรื่องนี้เป็นเรื่องยาว ผมจึงขอยกจุดเด่นอีกประการหนึ่งของระบบ GUIs นั่นคือ Hardware Independent มาพูดก่อน เริ่มเลยนะครับ

Hardware Independent

Hardware Independent หรือ Device Independent หมายความว่า

แอพพลิเคชันที่วิ่งอยู่บนฮาร์ดแวร์ประเภทหนึ่งสามารถที่จะโยกย้ายไปรันอยู่บนฮาร์ดแวร์อีกประเภทหนึ่งที่ตัวแอพพลิเคชันตัวนั้นสนับสนุน (support) แต่การไปวิ่งบนฮาร์ดแวร์ใด ๆ นั้น ต้องมีข้อกำหนดหรือสเปคที่ฮาร์ดแวร์นั้นสามารถทำได้ด้วย ประโยชน์ที่บรรดาโปรแกรมเมอร์ได้รับจากจุดเด่นข้อนี้ก็คือ ไม่ต้องคอยมาแก้ไขโปรแกรมให้ทำงานกับ คีย์บอร์ด, จอภาพ, เครื่องพิมพ์ หรือพลอตเตอร์ที่แตกต่างกัน เช่นการใช้โลจิคัลโคออร์ดิเนตในฟังก์ชันกราฟิกแล้วถูกแม็พปิงทับฟิสิคัลโคออร์ดิเนตเองโดยดีไวซ์ไดรเวอร์ของระบบ จากหัวข้อ GDI ที่ผ่านมาเราจะเห็นว่าโปรแกรมที่เขียนบน Window ไม่ได้แอดเซสตัวฮาร์ดแวร์ของกราฟิกดีไวซ์ไม่ว่าจะเป็นจอภาพหรือเครื่องพิมพ์โดยตรง แต่เราใช้ API (Application Programming Interface) กลุ่มหนึ่งที่ชื่อว่า GDI แทน Windows ก็เหมือนกับแอพพลิเคชันอื่น ๆ ที่ต้องอาศัยดีไวซ์ไดรเวอร์ในการเปลี่ยนผลทางกราฟิกจากโปรแกรมไปเป็นเอาต์พุตบนอุปกรณ์จริง ๆ ขอให้ดูรูปที่ 1 ประกอบ จะเห็นว่ามีดีไวซ์คอนเท็กซ์ ดีไวซ์ไดรเวอร์และเอาต์พุตดีไวซ์เข้ามาเกี่ยวข้อง สำหรับฟังก์ชันต่าง ๆ ของ GDI นั้นจะดึงมาจาก GDI.EXE กับ USER.EXE (หมายเหตุ : มีหลาย ๆ คนชอบที่จะกล่าวว่า Windows นั้น Device Independent Graphics Interface มากกว่าจะกล่าวกว้าง ๆ ว่า Device Independent ผู้อ่านคิดว่าอย่างไรล่ะครับ)

วิธีการที่ Windows ใช้ในการทำตัวเองให้มีคุณสมบัติทางด้าน hardware independent ก็คือการระบุลงไปอย่างแน่ชัดถึงคุณสมบัติขั้นพื้นฐานที่



รูปที่ 1 ลำดับการทำงานใน GUIs

ฮาร์ดแวร์แต่ละประเภทต้องสามารถทำได้ (เช่น พลอตเตอร์ต้องวาดเส้นตรงได้ทุกองศา) สิ่งเหล่านี้จะสร้างความมั่นใจให้แก่บรรดาผู้เขียนโปรแกรมว่า SDK รูทีนที่เขาใช้อยู่จะสามารถทำงานได้อย่างปกติบนฮาร์ดแวร์ใด ๆ ก็ตามที่มีคุณสมบัติพื้นฐานเหล่านี้อยู่ ดังนั้นบรรดาผู้ผลิตฮาร์ดแวร์รายใดก็ตามที่ปรารถนาจะให้ผลิตภัณฑ์ของตนใช้กับ Windows ได้นั้น ก็ต้องผลิตอุปกรณ์ให้มีความสามารถขั้นพื้นฐาน (minimum capabilities) ตามกำหนด สำหรับมาตรฐานต่าง ๆ นี้ได้ถูกกำหนดโดยทางบริษัทไมโครซอฟต์ไว้ใน Microsoft Windows

OEM Adaptation Kit ทุก ๆ รูทีนของ SDK ไม่ว่าจะเป็นฟังก์ชันพื้น ๆ หรือฟังก์ชันที่มีความซับซ้อนต่างก็มีความสามารถในการใช้ชุดคำสั่งขั้นพื้นฐานในการทำงานบนอุปกรณ์ได้ เช่นไม่ใช่ทุก ๆ พลอตเตอร์จะสามารถวาดวงรีได้ แต่โปรแกรมที่ใช้นั้นต้องเรียกใช้ฟังก์ชันในการวาดวงรีได้ ขอเพียงแต่พลอตเตอร์ที่นำมาต่อมีความสามารถขั้นพื้นฐานคือวาดเส้นตรงได้ ดังนั้นหากพลอตเตอร์ที่ใช้สามารถวาดวงรีได้ Windows ก็จะใช้ความสามารถพิเศษนั้น หากว่าไม่ Windows ก็จะใช้วิธีการวาดโดยใช้เส้นตรงสั้น ๆ เป็นจำนวนมากต่อกันเป็นวงรีแทน สรุปก็คือรูทีนทั้งหลายจะพยายามใช้ความสามารถของอุปกรณ์นั้นเพื่อทำให้การทำงานใช้เวลาน้อยที่สุดเท่าที่จะทำได้ นอกจากในแง่ของเอาต์พุตดีไวซ์แล้ว ในส่วนของอินพุตดีไวซ์ตัว Windows เองก็ได้ระบุถึงขีดความสามารถขั้นพื้นฐานหรือขั้นต่ำสุด (minimum capabilities) ของอินพุตดีไวซ์ที่จะทำให้ตัวแอพพลิเคชันของเราได้รับแต่อินพุตที่ถูกต้องคืออินพุตที่มีการกำหนดไว้ล่วงหน้าแล้ว (predefined input) อย่างในกรณีของการกดคีย์บอร์ด Windows ได้กำหนดชุดของการกดคีย์ที่มีไว้แล้ว (set of keystrokes) ซึ่งในการใช้งานแอพพลิเคชันจะไม่เคยได้รับ message ของคีย์บอร์ดนอกเหนือไปจากที่ Windows กำหนดไว้เลยและสำหรับ Windows ซึ่งเป็น GUIs บนเครื่องระดับพีซี บรรดาชุดของคีย์สโตรกที่เป็นไปได้ล้วนแล้วแต่มาจากการกดคีย์ทางคีย์บอร์ดในตระกูลไอบีเอ็มพีซีทั้งสิ้น ดังนั้นหากใครก็ตามที่ผลิตคีย์บอร์ดที่มีคีย์พิเศษนอกเหนือไปจากคีย์ที่ทาง Windows กำหนดไว้ (Windows virtual keyboard)

จะต้องมีซอฟต์แวร์ที่จะช่วยในการแปลงรหัสคีย์บอร์ดพิเศษเหล่านั้นไปเป็นคีย์สโตรกที่ Windows ยอมรับ สำหรับรูปที่ 2 แสดงตัวอย่างของ Windows Virtual Keyboard ผมจะขอยกตัวอย่างว่ามีคีย์บอร์ดรุ่นหนึ่งที่สร้างคีย์สำหรับเลื่อนเคอร์เซอร์ในแนวเฉียง ซึ่งในชุดของ Windows virtual keyboard ไม่มี ดังนั้นผู้ผลิตคีย์บอร์ดรุ่นนี้จะต้องทำซอฟต์แวร์สำหรับการแปลงไปเป็นโค้ดที่ Windows รู้จัก เช่นกดปุ่มเฉียงลงทางขวา ก็ต้องถูกแปลงเป็นกดปุ่มลูกศรทางขวาแล้วตามด้วยปุ่มลูกศรลง ขอย้ำอีกทีนะครับว่าบรรดาอินพุตทั้งหลายไม่ว่าจะจากคีย์บอร์ดหรือเมาส์ก็ตาม ล้วนแล้วแต่ถูกกำหนดไว้เรียบร้อยแล้ว แอพพลิเคชันที่สร้างขึ้นจึงไม่มีวันที่จะได้รับอินพุตนอกเหนือไปจากที่ถูกกำหนดไว้แล้วอย่างเด็ดขาด (never receive any undefined input) ดังนั้นถ้าเราใช้แอพพลิเคชันของเรากับเมาส์ที่มี 10 ปุ่ม (ผมยังนึกไม่ออกว่าจะเอาปุ่มมากมายอย่างนี้ไปทำอะไร) แอพพลิเคชันของเราก็ยังคงทำงานเหมือนกับใช้เมาส์ธรรมดา และมันก็จะไม่เคยได้รับเมาส์อินพุตนอกเหนือไปจากที่กำหนดไว้ จากความสามารถทั้งด้านอินพุตและเอาต์พุตที่กล่าวมา ทำให้บรรดาผู้พัฒนาโปรแกรมมีความรู้สึกว่าตนเองกำลังพัฒนาโปรแกรมให้ทำงานอยู่บนเครื่องที่เรียกว่า Windows Virtual Machine โดยอินพุตที่โปรแกรมสนใจก็จะเป็นอินพุตที่สามารถมีได้จริง ๆ (ขอเพียงอยู่ในเซตของ predefined input) และเอาต์พุตรูทีนก็สามารถทำงานได้จริง ๆ โดยไม่ต้องคอยพะวงกับฮาร์ดแวร์ที่ใช้ (ขอเพียงแต่เป็นดีไวซ์ที่สามารถทำงานบน Windows ได้ คือทำตามข้อกำหนดพื้นฐานได้)

```
WINDOWS.H
Identifier              IBM Keyboard

VK_CANCEL               Ctrl-Break
VK_BACK                 Backspace
VK_RETURN               Enter
VK_MENU                 Alt
VK_PRIOR                Page Up
VK_NEXT                 Page Down
VK_NUMPAD0              Numeric keypad 0 with Num Lock ON
```

รูปที่ 2 แสดงบางส่วนของ VIRTUAL KEYBOARD ที่ WINDOWS กำหนดไว้แล้ว

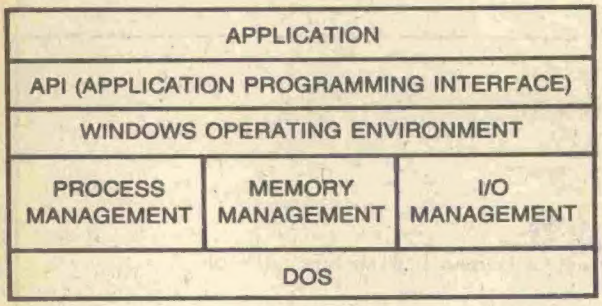
แต่ในโลกนี้ไม่มีอะไรที่ได้มาง่าย ๆ หรอกครับ การเขียนโปรแกรมบน Windows หรือ GUIs ตัวอื่น ๆ ล้วนแล้วแต่ต้องใช้เวลาในการเรียนรู้สูงทั้งสิ้น การที่เราจะรู้ว่าเมื่อไรควรจะใช้ near, far, short, long, local หรือ global ล้วนแล้วแต่ต้องใช้เวลาทั้งสิ้น ซึ่งเรื่องนี้จะไว้พูดกันในช่วงที่สองอันว่าด้วยเทคนิคการเขียนโปรแกรมกันอีกทีหนึ่ง

ทำไมต้อง Message

โปรแกรมที่เขียนเพื่อทำงานบน DOS โดยทั่วไปจะสมมติตัวเองว่าสามารถเข้าไปครอบครองการใช้ทรัพยากรต่าง ๆ ในระบบทั้งอุปกรณ์ทางด้าน input, output หน่วยความจำ จอภาพและตัวโปรเซสเซอร์เอง แต่ในโปรแกรมที่เขียนสำหรับทำงานภายใต้ Windows หรือ GUIs ตัวอื่น ๆ กลับต้องใช้ทรัพยากรอันมีค่าของระบบเหล่านี้รวมกันกับโปรแกรมอื่น ๆ ที่ทำงานอยู่ในขณะเดียวกัน ในระบบปฏิบัติการทั่ว ๆ ไปที่เป็นระบบที่มีความสามารถทางมัลติทาสกิง ตารางเวลาการใช้ซีพียูได้ถูกกำหนดโดย clock ในระบบ แต่ละโปรแกรมจะมีช่วงเวลาหรือ time slice ของมันเอง เมื่อหมดช่วงเวลา มันก็จะถูกขัดจังหวะ (interrupt) และหยุดการทำงานเพื่อให้โปรแกรมตัวอื่นเข้ามาทำงาน (เข้ามามีโอกาสใช้ซีพียูบ้าง) เราเรียกระบบนี้ว่าเป็นระบบ preemptive scheduling (หมายเหตุ : ถ้าจะว่ากันตามหลักวิชา OS แล้วควรใช้คำว่าโปรเซสมากกว่าโปรแกรม) ในระบบ preemptive scheduling โปรแกรมถูกขัดจังหวะการทำงานโดยตัวระบบปฏิบัติการสำหรับ Windows แล้วเป็นระบบ nonpreemptive scheduling บรรดาแอพพลิเคชันบน Windows ไม่ได้ถูกขัดจังหวะโดย OS หากแต่มันขัดจังหวะตัวเอง โดยอาศัยกลไกของเมสเสจคือเมื่อมันไม่มีเมสเสจก็จะให้แอพพลิเคชันอื่นเข้ามาใช้ซีพียูแทนโดยอัตโนมัติ เพื่อให้ได้ความสามารถในการทำมัลติทาสกิง Windows จึงไม่สามารถปล่อยปละละเลยให้โปรแกรมแต่ละตัวเข้าไปควบคุมการใช้ฮาร์ดแวร์ได้โดยตรงเพราะจะสร้างความปั่นป่วนให้แก่ระบบ นั่นคือ Windows ต้องเข้าไปควบคุมฮาร์ดแวร์เองซึ่งเราอาจมองการเข้าไปควบคุมฮาร์ดแวร์ของ Windows ได้เป็นกลุ่ม ๆ ดังนี้

– เข้าไปควบคุมอินพุตจากผู้ใช้ทั้งหมดและแจกจ่ายไปในรูปของเมสเสจให้แก่แอพพลิเคชันที่เหมาะสมต่อไป

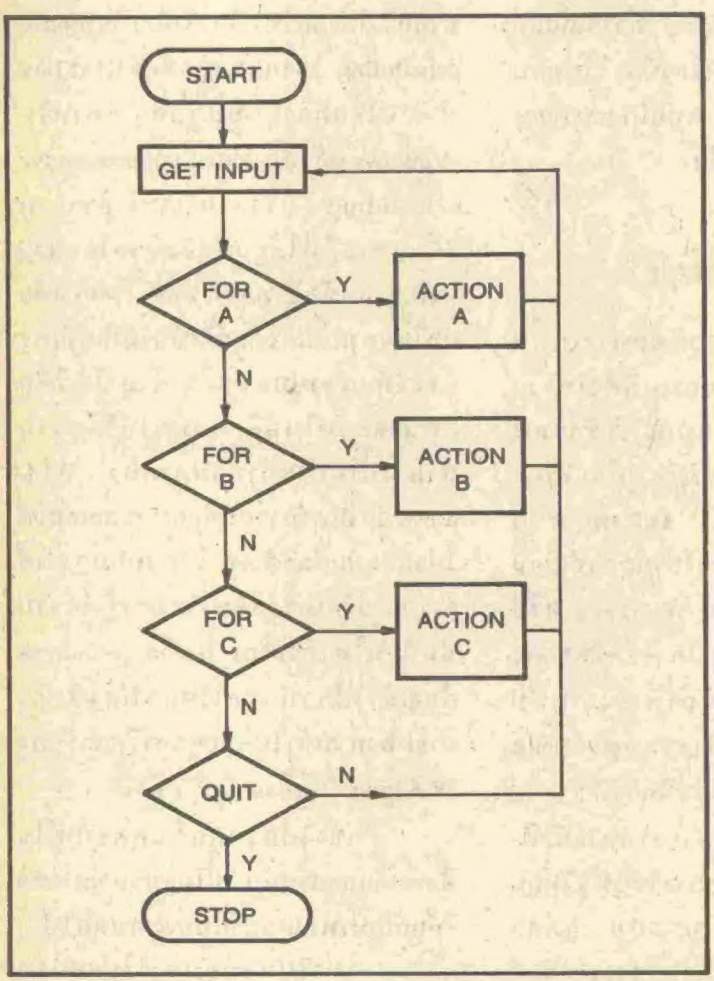
– เข้าไปควบคุมการใช้จอภาพของแต่ละวินโดว์ (แต่ละแอพพลิเคชันอาจมีได้มากกว่าหนึ่งวินโดว์)



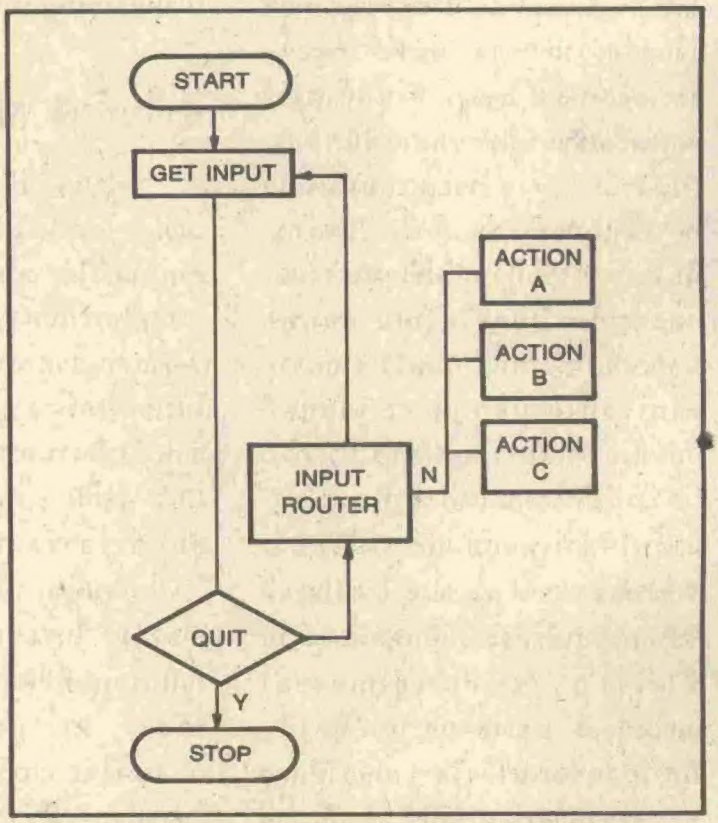
รูปที่ 3 แสดงโครงสร้างของระบบที่ใช้ WINDOWS

– เข้าไปควบคุมการใช้เอาต์พุตดีไวซ์ของแต่ละแอพพลิเคชัน

จากการที่ Windows เข้าไปควบคุมฮาร์ดแวร์ ทำให้การติดต่อระหว่างฮาร์ดแวร์ในแอพพลิเคชันต้องผ่านตัว Windows เสมอ เช่นวินโดว์แอพพลิเคชันจะไม่เขียนข้อความโดยตรงหากแต่ใช้ GDI รูทีนแทน ไม่ทำฮาร์ดแวร์อินเตอร์-รัพต์เอง หากแต่ใช้การดักเมสเสจที่เหมาะสมแทน ไม่ส่งข้อมูลลงไปพิมพ์ที่พรินเตอร์โดยตรง ในรูปที่ 3 จะแสดงลำดับชั้นการทำงานของระบบ โดยเราจะเห็นว่าส่วนที่ทาง Windows ไม่เข้าไปแทรกแซงดอสก็คือส่วนจัดการไฟล์ (File Management) โปรแกรมเมอร์โดยส่วนใหญ่จะคุ้นเคยกับการเขียนโปรแกรมในแบบซีเควนเชียลซึ่งมีการทำงานเป็นขั้นเป็นตอนตั้งแต่ต้นจนจบ และรูปแบบการเขียนโปรแกรมแบบนี้ต่างกับรูปแบบการเขียนโปรแกรมที่เป็นแบบเมสเสจไดรเวน (message-driven หรือ event-driven) ดังแสดงในรูปที่ 4 และ 5 เพราะฉะนั้นจะเห็นได้ว่าในการโปรแกรมแบบเมสเสจไดรเวนนั้น ส่วนที่โปรแกรมต้องคอยสนใจก็คือการพิจารณาว่าเมสเสจใดที่โปรแกรมควรจะสนใจ และเมสเสจใดที่ไม่ต้องสนใจโดยเมื่อมีเมสเสจที่สนใจเกิดขึ้น แอพพลิเคชันจะต้องทำการพิจารณาว่าจะต้องทำอะไร



รูปที่ 4 โครงสร้างโปรแกรมบน Sequential แต่ละ ACTION ไม่สามารถทำงานพร้อมกันได้ เพราะมี INPUT เข้ามาทีละแบบ



รูปที่ 5 โครงสร้างโปรแกรมแบบ message-driven แต่ละส่วนสามารถทำงานไปพร้อม ๆ กันได้เพราะไม่ต้องคอย INPUT

บ้างตามที่โปรแกรมเขียนไว้ เมื่อทำเสร็จก็คอยเมสเสจต่อไป การที่มีเมสเสจเกิดขึ้นก็คือการที่เกิดอีเวนต์ขึ้นในระบบเช่น มีคีย์บอร์ดอีเวนต์ (การกดปุ่มคีย์บอร์ด) เมาส์อีเวนต์ (การลากเมาส์การกดปุ่มเมาส์) อีเวนต์เหล่านี้จะผ่าน Windows และถูกแปลงเป็นเมสเสจส่งไปให้แก่แอพพลิเคชันอีกทีหนึ่ง (แต่ก็มีบางทีที่มีเมสเสจเกิดขึ้นโดยไม่มีอีเวนต์ เช่น เมื่อแอพพลิเคชัน A ส่งเมสเสจไปให้แอพพลิเคชัน B)

โครงสร้างของ Message

จากที่ผ่านมาเรามักพูดถึงในแง่ของเมสเสจที่ส่งมาให้แต่ละแอพพลิเคชัน แต่หากจะพูดกันจริง ๆ แล้วเมสเสจนั้นมีการระบุลงไปอย่างละเอียดว่าเป็นเมสเสจของวินโดว์ใด (อย่าลืมว่าแต่ละแอพพลิเคชันสามารถมีได้หลายวินโดว์พร้อม ๆ กัน) ทุก ๆ เมสเสจไม่ว่าจะเป็นประเภทใดล้วนแล้วแต่ประกอบด้วยโครงสร้างเหมือนกันดังในรูปที่ 6 หากแต่ที่น่าสนใจจะมีเพียง 4 ฟิลด์คือ

- handle ของวินโดว์ขนาด 16 บิต
- message type ขนาด 16 บิต
- WORD parameter ขนาด 16 บิต
- LONG parameter ขนาด 32 บิต

handle ของวินโดว์จะเป็นค่า ID ที่ใช้ในการอ้างถึงวินโดว์ที่จะเป็นเจ้าของเมสเสจนี้ (เป็นผู้ได้รับเมสเสจนี้) ส่วน message type เป็นค่าตัวเลขที่ระบุถึงประเภทของเมสเสจที่ส่งมา โดย Windows ได้ตั้งชื่อให้เมสเสจแต่ละประเภทไว้ โดยประกอบด้วยตัวอักษรย่อสองตัว ตามด้วยเครื่องหมาย underscore และลงท้ายด้วยกลุ่มสตริงเช่น WM_CLOSE, LB_DELETESTRING โดยในที่นี้

```
typedef struct tagMSG {
    HWND   hwnd;
    WORD   message;
    WORD   wParam;
    LONG   lParam;
    DWORD  time;
    POINT  pt;
} MSG;
```

รูปที่ 6 แสดงโครงสร้างของข้อมูลชนิด message

WM หมายถึง Window message และ LB หมายถึง List Box message อีก 2 ฟิลด์ที่เหลือเป็นค่าพารามิเตอร์ในขณะนั้น ส่วนจะเป็นค่าอะไรก็ต้องแล้วแต่ว่าเมสเสจที่ส่งมานั้นเป็นประเภทอะไร โดยมักจะเก็บค่าพวกตำแหน่งของเมาส์ ตำแหน่งของ scroll bar คีย์บอร์ดที่ถูกกด ฯลฯ

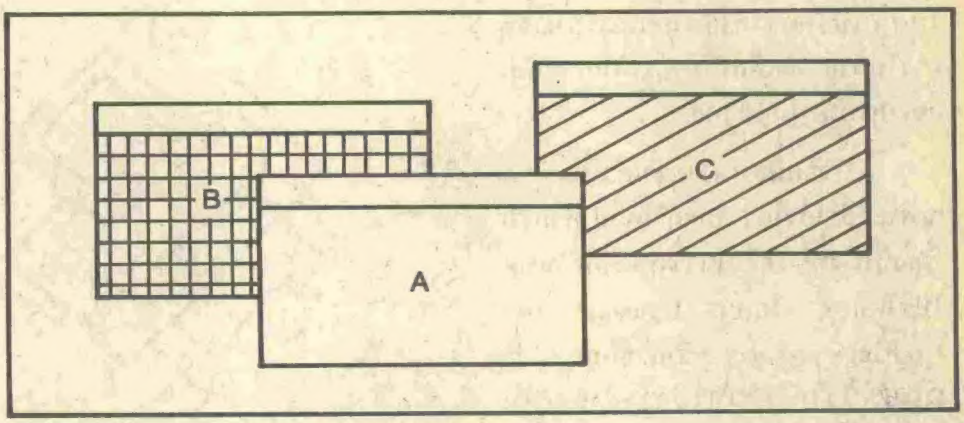
ที่มาของ Message

เรารู้ว่า Windows ใช้กลไกของเมสเสจในการควบคุมการทำงานแบบมัลติทาสกิงดังนั้น (เกือบ) ทุก ๆ เมสเสจจะถูกส่งผ่าน Windows แต่นั้นก็ไม่ได้หมายความว่า Windows จะเป็นผู้สร้างเมสเสจทุก ๆ เมสเมจที่มีในระบบ หากแต่เมสเสจทั้งหลายมีต้นกำเนิดมาจากที่ต่าง ๆ ในระบบ 4 จุดด้วยกันคือ

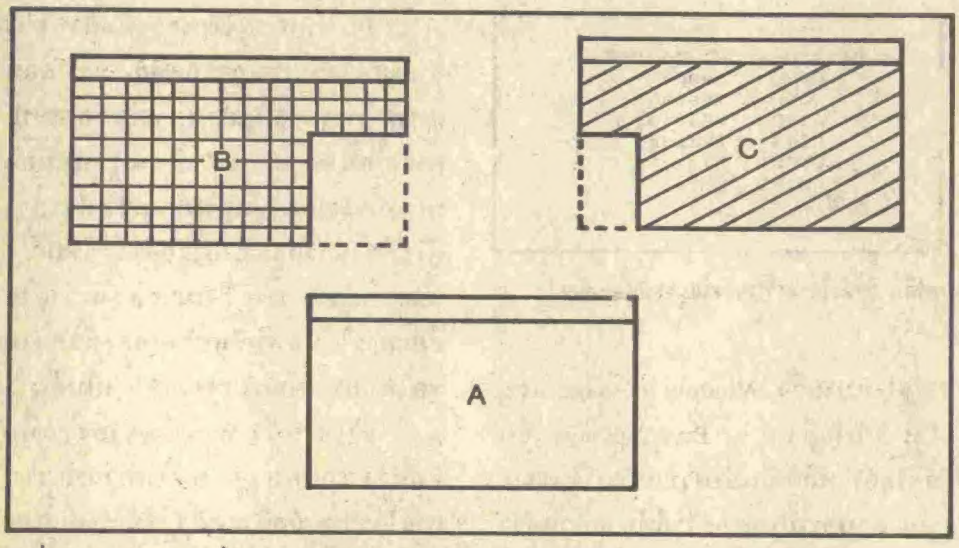
1) จากตัวผู้ใช้เอง คือผ่านทางอินพุตดีไวซ์ เช่นการกดคีย์, การเลื่อนเมาส์, การกดปุ่มเมาส์, การเลือกเมนู, การเลื่อน scroll bar เป็นต้น เมสเสจเหล่านี้จะบอกให้แอพพลิเคชันทราบว่าผู้ใช้ต้องการกระทำบางอย่าง เช่นเมื่อเลื่อน scroll bar ก็หมายความว่าผู้ใช้ต้องการให้แอพพลิเคชันแสดงข้อความชุดใหม่แทนข้อความชุดเดิม เป็นต้น

2) จากตัว Windows เอง เพื่อจัดการสภาพแวดล้อมในการทำงานภายใต้ระบบมัลติทาสกิง เช่น ส่งเมสเสจให้แอพพลิเคชันวาดบริเวณไคลเอนต์-แอเรียใหม่เมื่อวินโดว์ที่เคยแสดงทับอยู่ถูกนำออกไปแล้ว ดังแสดงในรูปที่ 7, 8 และ 9

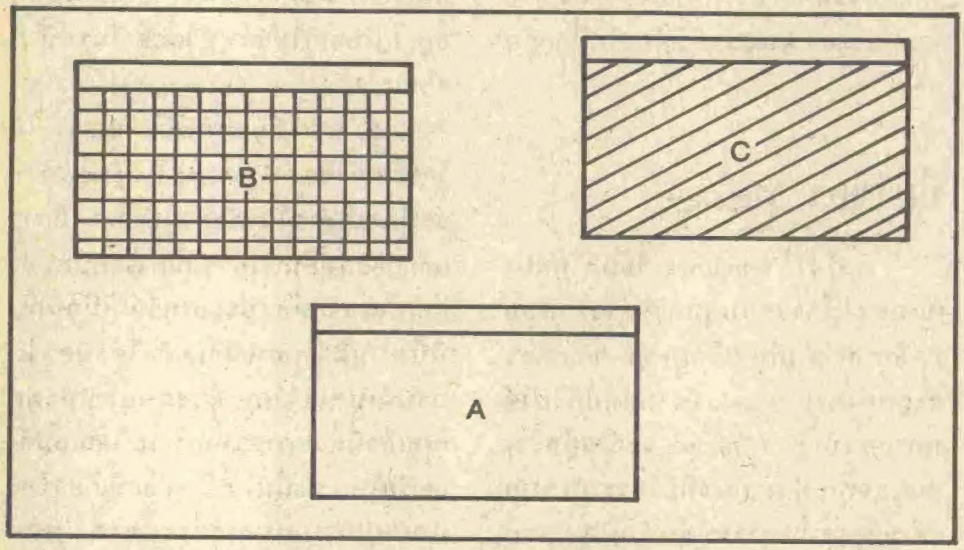
3) จากตัวแอพพลิเคชันเอง ในโครงสร้างของ Windows Program ดังที่จะมีการกล่าวในครั้งต่อไปนั้น มีการแตกย่อยลงเป็นส่วน ๆ ในการตอบสนองต่อเมสเสจแต่ละประเภทไม่เหมือนกัน ที่นี้สมมติว่าแอพพลิเคชันได้รับคอมมานด์เมสเสจและต้องตอบสนองต่อเมสเสจนี้ด้วยการวาดบริเวณไคลเอนต์-แอเรียใหม่ ดังนั้นในส่วนย่อยที่เกี่ยวข้องกับคอมมานด์เมสเสจจะต้องส่งเพนต์เมสเสจ (paint message) ออกไป เพื่อ



รูปที่ 7 WINDOW A ทับ WINDOW B และ C อยู่



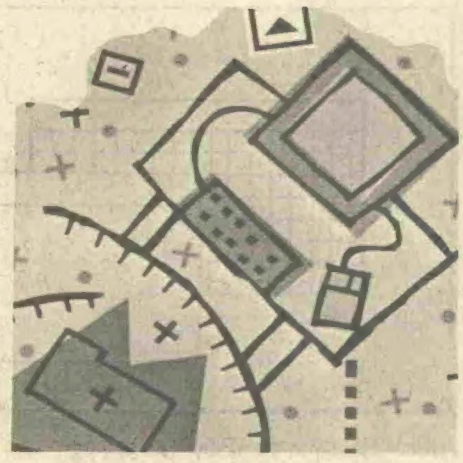
รูปที่ 8 WINDOW A ถูกเคลื่อนย้ายออกไป เกิดบริเวณว่างในบางส่วน CLIENT AREA ของ WINDOW



รูปที่ 9 WINDOW B และ C ได้รับ PAINT MESSAGE ที่ WINDOWS ส่งมา จึงทำการวาด CLIENT AREA ใหม่

ให้ส่วนที่จัดการกับเพนต์เมสรับมาจัดการในภายหลังเพื่อทำการวาดไคลเอนต์-แอเรียใหม่ตามต้องการ

4) จากแอพพลิเคชันตัวอื่น โดยเฉพาะอย่างยิ่งในระบบเครือข่าย สำหรับเรื่องนี้ทางบริษัทไมโครซอฟต์ได้กำหนดโปรโตคอล เรียกว่า Dynamic Data Exchange protocol หรือเรียกย่อ ๆ ว่า DDE ไว้ให้ใช้สำหรับการส่งและรับเมสเสจระหว่างกัน



ประเภทของ message ที่พบเห็นบ่อย ๆ

- Create message

ทุกครั้งที่วินโดว์ของแอพพลิเคชันถูกสร้างขึ้น จะมีคลีเอตเมสเสจส่งมาให้เพื่อเป็นการบอกว่ามีวินโดว์ใหม่ถูกสร้างขึ้น ซึ่งมักจะมีการดักเมสเสจนี้เพื่อทำการเซตค่าเริ่มต้นตามที่ต้องการ ทุก ๆ วินโดว์ที่ถูกสร้างขึ้นมาจะได้รับเมสเสจนี้เพียงครั้งเดียว

- Destroy message

เมสเสจประเภทนี้ตรงกันข้ามกับคลีเอตเมสเสจคือมันจะถูกส่งไปให้ก็ต่อเมื่อมีการปิดวินโดว์ (ถูกลบออกไปจากจอภาพ) เรียบร้อยแล้ว ดังนั้นแต่ละวินโดว์จะได้รับเมสเสจนี้เพียงครั้งเดียวในช่วงชีวิตของมันเช่นเดียวกัน ในกรณีที่แอพพลิเคชันมีพาเรนต์วินโดว์ (parent window) และไชลด์วินโดว์ (child window) เมื่อเลิกการทำงานของแอพพลิเคชันไชลด์วินโดว์จะต้องถูกปิดตัวเอง โดยอาศัย destroy message ซึ่งอาจถูกส่งมาจากพาเรนต์วินโดว์หรือจาก Windows เอง

- Close message

เมสเสจนี้เป็นการบอกให้ทราบว่าผู้ใช้มีความต้องการที่จะปิดวินโดว์ซึ่งโดยส่วนใหญ่แล้วตามปกติจะทำให้เกิด destroy message ตามมา (เมื่อแอพพลิเคชันได้รับ close message แล้วไม่สนใจเมสเสจนี้ก็จะถูกส่งต่อไปให้ Windows เป็นผู้พิจารณา โดยใช้ฟังก์ชัน DefWindowProc ซึ่งจะทำให้เกิด destroy message ตามมา) ตามปกติแล้วในแอพพลิเคชันที่เมื่อมีการปิดวินโดว์แล้วจะทำให้ไฟล์เสียหรือข้อมูลที่ทำอยู่สูญหายได้นั้นมักมีการเปิด message box ขึ้นมา

เตือนผู้ใช้งานเสมอ ดังที่เห็นในรูปที่ 10 และ 11 ซึ่งวิธีการนี้สามารถทำได้โดยการดัก close message นี้ไว้ (ซึ่งจะเป็นการทำไม่ให้เกิด destroy message ตามปกติ) จะเห็นได้ว่า clolse message นั้นเป็นเพียงแต่บอกเราว่ามีความต้องการที่จะปิดเท่านั้น แต่ยังไม่มีการปิดจริง ๆ เกิดขึ้น ส่วน destroy message นั้นกลับบอกเรามาได้มีการปิดวินโดว์ไปแล้ว

– Quit message

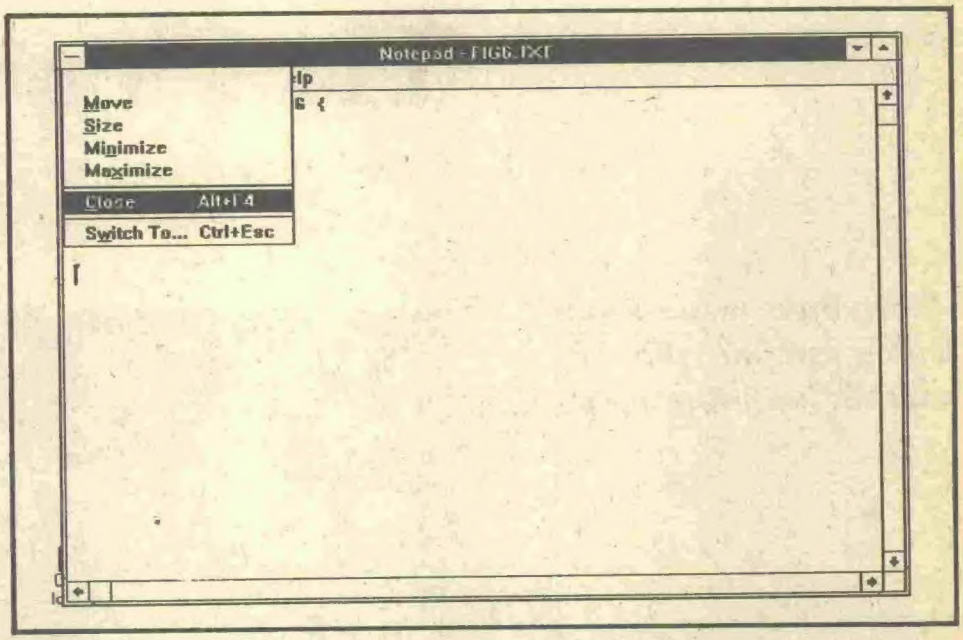
quit message ก็คล้าย ๆ กับ destroy message หากแต่ว่า destroy message นั้นเป็นการปิดแค่เพียงหนึ่งวินโดว์ในขณะที่ quit message เป็นการปิดทุกวินโดว์ของแอพพลิเคชัน (ยุติการทำงานของแอพพลิเคชัน)

– Command message

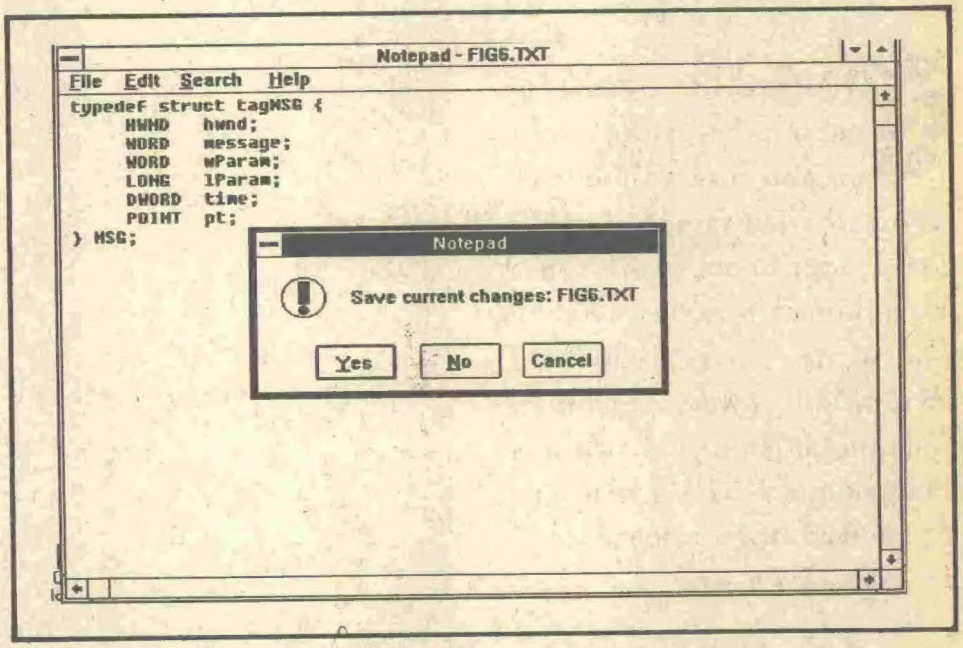
command message เป็นการบอกให้เราทราบว่ามีรีเควสต์ (request) บางอย่างจากผู้ใช้เกิดขึ้น เช่น เมื่อมีการเลือกเมนู ก็จะมีการส่ง command message มาให้เรา ซึ่งเราสามารถตรวจสอบได้ว่าเป็นการเลือกรายการใดในเมนูจากค่าพารามิเตอร์ของเมสเสจ (ค่า wParam, lParam)

– Paint message

message นี้ดูจะมีความสำคัญที่สุดก็ว่าได้ โดยมันมีหน้าที่อยู่ 2 ประการคือ บอกให้แอพพลิเคชันทราบว่าสิ่งที่กำลังแสดงอยู่บนไคลเอนต์แอเรีย ไม่สอดคล้องกับข้อมูลปัจจุบันและต้องมีการวาดบริเวณไคลเอนต์แอเรียใหม่ ตามปกติแอพพลิเคชันจะเป็นผู้ส่ง paint message นี้ออกไปเอง เช่นมีการรับข้อความจากผู้ใช้ผ่าน text entry box และต้องมีการนำไปแสดงลงบนไคลเอนต์แอเรีย แต่บางที paint message ก็อาจถูกส่งมาจาก Windows เองก็ได้ เช่นมีแอพพลิเคชันอื่นเปิด popup window ทับบางส่วนของไคลเอนต์แอเรียของแอพพลิเคชันอีกตัวหนึ่ง เมื่อมีการนำ popup window ออกไปแล้ว Windows จะส่ง paint message มาบอกแก่แอพพลิเคชันที่ถูกทับเพื่อให้แอพพลิเคชันตัวนั้นทำการวาดบริเวณไคลเอนต์แอเรียใหม่ให้ถูกต้อง



รูปที่ 10 ผู้ใช้เลือกรายการ CLOSE จาก SYSTEM MENU ซึ่งจะทำให้เกิด CLOSE MESSAGE ไปยังแอพพลิเคชัน



รูปที่ 11 แอพพลิเคชันดัก CLOSE MESSAGE ไว้ พร้อมกับแสดงการทำงานบางอย่างออกมา

ตอนนี้ผมชักรู้สึกว่าจะพูดเรื่องนี้ยาวไปเสียแล้ว คงต้องขอค้างไว้แค่นี้ก่อนนะครับ เดี๋ยวทั้งผู้อ่านและผมจะปวดหัวแย่ ไว้ฉบับหน้าเราจะมาพูดถึงเรื่องนี้กันให้จบ สวัสดีครับ